Heartbleed: Lessons from the System Failure

Christof Leng

International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, CA 94704, USA

Abstract: The Heartbleed bug in OpenSSL has both be described as one of the worst security vulnerabilities in the history of the internet and as a simple programming error. I argue that vulnerabilities like Heartbleed are not a coincidence or inevitable, but a result of the overall process underlying the design and implementation of current internet protocols. In order to minimize the probability of similarly catastrophic failures in the future, many aspects of the protocol engineering culture need to be revisited. Since the internet has become a fundamental infrastructure for almost every aspect of our society, it is crucial to address these issues and to (re-)establish trust in its security and reliability.

1 Introduction

On April 7th 2014, news about the discovery of the Heartbleed bug hit mainstream media worldwide. A rather trivial missing check in the Heartbeat extension of the popular OpenSSL library potentially compromised hundreds of thousands of servers worldwide and might have exposed user data, passwords, and even private encryption keys of millions of users and websites.

OpenSSL is the most popular open source implementation of the internet transport security protocols SSL, TLS, and DTLS, and is part of many Linux distributions, network appliances like home routers, and a large fraction of HTTPS-secured websites. The Secure Socket Layer (SSL) [Hic95] was an attempt to add security to TCP-based application protocols like HTTP, SMTP, or FTP. It later evolved into Transport Layer Security (TLS) [Die08], which currently is available as version 1.2, and turns the application protocols into their secured counterparts HTTPS, SMTPS, and FTPS. Datagram Transport Layer Security (DTLS) [RM12] is a version of TLS that provides security for connectionless application protocols that are based on UDP or similar transport protocols. It is used by some Virtual Private Network (VPN) software or the WebRTC protocol for browser-to-browser real-time communication, among others.

It is not the intention of this paper to blame any individual, project, or organization for the Heartbleed bug or any circumstance that may have promoted it. Anyone who dedicates time to the design, implementation, and operation of the internet deserves our deepest respect, especially if the contributions are made in the form of open standards or free software. Instead, this discussion is an attempt to highlight potential shortcomings of the status quo and to make suggestions on how to improve the situation in the common interest.

2 Background

In order to understand the Heartbleed bug and the various mechanisms that may have promoted it, we first need to understand TLS/DTLS and its heartbeat extension.

The OpenSSL project dates back to 1998 and is based on the earlier SSLeay implementation. The SSL protocol was first developed by Netscape and publicly released as version 2.0 and 3.0. It was later standardized by the Internet Engineering Task Force (IETF) under the name TLS. The IETF publishes its standard documents under the name request for comment (RFC). The current version of TLS is 1.2. The TLS protocol is extendable and numerous extensions exist, which are also defined in RFCs. DTLS re-uses most of the TLS structure and only modifies it where necessary to preserve the unreliable datagram semantics of the underlying transport.

Both protocols use a record structure to encapsulate the application protocol messages and their own control messages. The header of the record layer consists of a content type field, a protocol version number, and a length field (see Figure 1). DTLS adds a sequence number and an epoch field. In the remainder of the paper, TLS record is used to describe both DTLS and TLS records. The type field is a protocol discriminator, which determines how the content of the record has to be processed.

2.1 Heartbeat Use-Cases

The heartbeat extension defines one of the currently five TLS record content types and was defined in RFC 6520 [TSW12]. It has multiple use cases, among them keep-alive, path MTU discovery, and mobility support.

Keep-alive is a mechanism to detect unresponsive communication partners. A host sends a keep-alive request to its communication partner, and this receiver replies with a keep-alive response by echoing the payload of the request. The sender typically sends a counter value, pointing to a local table entry to keep responses apart.

Path MTU Discovery (PMTUD) is used to determine the maximum transfer unit (MTU) on the communication path between the two hosts. The MTU is the maximum message size intermediate routers can forward to the destination. PMTUD sends a series of probe packets to pinpoint the MTU. Standard-compliant routers unable to forward such a packet will respond with a “Fragmentation Needed” ICMP message and its MTU. The sender adjusts the probe packet size accordingly and the process is repeated until the destination is reached, which will notify the sender of the success. The heartbeat extension uses a padding field with random data to generate probe packets of the required sizes. The payload, which is echoed back upon success, is used to determine the path MTU.

The heartbeat extension was designed to accommodate future uses like mobility support. In modern internet use cases, hosts may change their public IP address at any time, often even without being able to detect the change themselves, because an intermediate middlebox (e.g., a NAT gateway) is responsible for the change. In this case, the connection breaks, because the packets can no longer be routed to the correct receiver. Heartbeats can help in two ways. First of all, a client that periodically sends heartbeat messages to the server, such that the server can detect the changed sender address even if the client sends no other data. The server then has to check the authenticity of the client. It sends a special heartbeat message as a challenge to the client. Only if the client is able to decrypt this message and echo the payload, the server sends future messages to the new address. In order to avoid resource attacks with fake client heartbeats, the server encodes all necessary information to process the reply in its heartbeat request. Mobility support in DTLS require a modified record format to identify the connection despite the address change. Therefore, it currently is only a theoretical concept.

2.2 Heartbeat Design

Heartbeat requests and replies for all different use-cases share the same message format, which is a superset of the required functionality for each scenario. The content type of this message format is 24. The message contains a one byte type field, which distinguishes between requests and replies (see Figure 1). It also contains a variable length payload field and a variable length padding field. The length of the payload field is determined by a two byte payload_length field. The length of the padding field can only be determined by subtracting the length of the other fields from the length field in the TLS record header. The formula is $padding\_length = length - 1 - 2 - payload\_length$. Additionally, the RFC requires the padding_length to be at least 16.

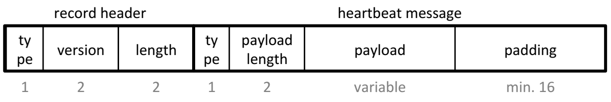


Figure 1: Heartbeat message format

In order to detect malformed heartbeat requests, the receiver must check that the message has at least the minimum length of 1 + 2 + 16, and that the content of the payload_length field is consistent with the total length of the record. Thus, $payload\_length \leq length - 1 - 2 - 16$. If the latter check is missing, a malicious sender may send a payload_length field much larger than the actual record size. The receiver then not only copies the actual payload into the heartbeat response, but also the memory content adjacent to it. By sending a very small heartbeat request containing no payload or padding at all, but a payload_length of the maximum value of 65535, the attacker can receive almost 64KB of memory content from the attacked. This is exactly how the Heartbleed attack works.

This is a very severe vulnerability, because it returns memory from more or less random locations, and repeated attacks can reveal critical content like passwords or private keys with a high probability. Additionally, the attack does not trigger any error, log message, or modify the attacked system's state and therefore leaves no trace at all.

The faulty heartbeat implementation in OpenSSL was accepted by the project maintainer on December 31st 2012 and has only been detected more than two years later. It is practically impossible to determine if it has been known to and used by attackers before its public disclosure.

3 Causes of Heartbleed

Superficially, Heartbleed is caused by the missing of a trivial security check that has been overlooked by the programmer and the reviewer. On the one hand, this may sound comforting, because it implies that there is nothing fundamentally wrong. On the other hand, because human error is inevitable, we must anticipate future bugs with similar or worse consequences. In the following, we will investigate a wide range of circumstances that have facilitated the bug and need to be addressed to reduce the probability of future vulnerabilities.

3.1 Programming Methods

The most obvious problem with the programming side is that Hearbleed is a buffer over-read bug. A buffer over-read is a type of buffer overrun, where data is read beyond the actual boundaries of the buffer. This is a problem known for more than 40 years [And72] and has well-known remedies. There is no excuse for buffer overruns in modern security-critical software, but they are still very frequent, because of the choice of **programming language**. C/C++ is probably the only still commonly used language that does not use automatic bounds checking by default. Unfortunately, C/C++ is especially dominant in systems programming and communication protocols.

Even with a language like C/C++, memory bugs can be minimized when the proper **tools** are used. Plain pointer arithmetic is more prone to buffer overruns than specialized data structures with internal boundary checks. Debugging tools like Valgrind [NS07] can help to discover memory bugs. Unfortunately, such tools would not work well with OpenSSL, because it uses (and abuses) its own custom memory management instead of using standard memory allocation [Una14]. OpenSSL also does not use comprehensive unit testing as a tool for automated quality assurance. In the aftermath of Heartbleed, an initiative was started to add more unit tests to OpenSSL [Bla14].

The challenge is aggravated by the **complexity** of the OpenSSL codebase. Currently, it consists of more than 450,000 lines of code [Ohl14]. That is more than five times the size of Google's QUIC, an experimental protocol that combines transport functionality with transport security. OpenSSL can only be blamed in part for this complexity, because SSL/TLS itself is a very complex standard (see next Section), which requires a lot of code to implement. Nonetheless, the LibReSSL fork [Bec14] managed to remove 150,000 lines of content from the code base within a few weeks.

With an unsafe programming style and without a proper tool chain for automated tests, the manual quality assurance becomes the crucial part of the process. The OpenSSL does manual **code reviews** of submitted source code contributions. The heartbeat extension did pass that check. Code reviews are an important tool for quality assurance, but can only reduce the probability of programming errors slipping through. Eric S. Raymond's so-called Linus's Law states that "given enough eyeballs, all bugs are shallow", which may already be debatable, but in the case of OpenSSL there were definitely not enough eyeballs. Despite being used all over the internet by many companies, governments, and enthusiasts, the project has been notoriously underfunded and thus only one developer could afford to work full-time on the project. A few weeks after Heartbleed, the Linux Foundation has formed a Core Infrastructure Initiative of IT companies providing substantial funding to OpenSSL and similar projects to improve the situation [The14]. Projects like OpenSSL seem to suffer from the "tragedy of the commons": Even though the software is used by millions or billions of users, countless companies, and more than a few governments, only very few individuals take responsibility and contribute money or time and simply rely on the work of a few altruistic volunteers.

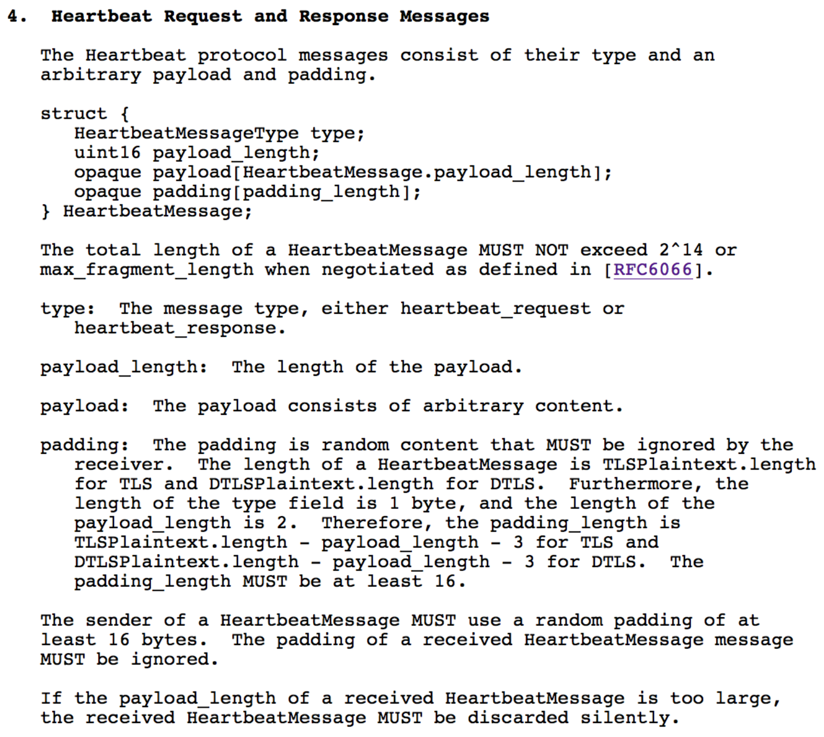
Given the circumstances, it must be rated as a combination of the effort of these individuals and pure luck that not more vulnerabilities of this magnitude have surfaced. Without a change in system programming culture and responsibility culture, the outlook is bleak.

3.2 Design Process

Heartbleed is not simply a programming problem. It roots much deeper into the design process of TLS and DTLS. First of all and most fundamentally, the whole concept of TLS vs. DTLS must put into question. As explained earlier, DTLS shares many concepts, formats, and even implementations with TLS. The goal of this approach was to minimize the overall effort and the ability to solve problems once instead of twice. Instead, a bug in the heartbeats, which only make sense for DTLS, affected the much more widespread and critical TLS. TLS's underlying transport, TCP, already provides perfectly fine PMTUD and keep-alive functionality. The mobility mechanism that could be built on top of heartbeats would not even work with TCP, because TCP connections cannot be redirected to a new IP address like this.

Even worse, DTLS only needs a heartbeat mechanism because it introduces concepts to connection-less datagram protocols which are intrinsic to connection-oriented stream protocols. A stateless UDP-based application protocol does not need a keep-alive, because there is no connection. But DTLS introduces a handshake to establish symmetric encryption and has to maintain the associated state. Thus, it needs to know, when this state can be thrown away. UDP-based application protocols also need to avoid packet fragmentation, because fragmented packets negatively impact the loss probability. Either the application protocol provides its own PMTUD or it avoids large datagrams altogether. Yet, the "magic" DTLS layer does introduce large messages itself, e.g., the certificate exchange in the connection handshake. Trying to make two things look the same, which are fundamentally different, is dangerous **over-generalization**.

This conceptual problem becomes evident through **contradictions** in the standards document. While RFC 6520 states heartbeats are meant as "a basis for path MTU (PMTU) discovery for DTLS" [TSW12], the DTLS 1.2 specification [RM12] specifically claims "DTLS's philosophy is to leave PMTU discovery to the application". The publication of the two RFCs is just one month apart, both have been in the standardization process in parallel.



```
4.  Heartbeat Request and Response Messages

   The Heartbeat protocol messages consist of their type and an
   arbitrary payload and padding.

   struct {
      HeartbeatMessageType type;
      uint16 payload_length;
      opaque payload[HeartbeatMessage.payload_length];
      opaque padding[padding_length];
   } HeartbeatMessage;

   The total length of a HeartbeatMessage MUST NOT exceed 2^14 or
   max_fragment_length when negotiated as defined in [RFC6066].

   type:  The message type, either heartbeat_request or
      heartbeat_response.

   payload_length:  The length of the payload.

   payload:  The payload consists of arbitrary content.

   padding:  The padding is random content that MUST be ignored by the
      receiver.  The length of a HeartbeatMessage is TLSPlaintext.length
      for TLS and DTLSPlaintext.length for DTLS.  Furthermore, the
      length of the type field is 1 byte, and the length of the
      payload_length is 2.  Therefore, the padding_length is
      TLSPlaintext.length - payload_length - 3 for TLS and
      DTLSPlaintext.length - payload_length - 3 for DTLS.  The
      padding_length MUST be at least 16.

   The sender of a HeartbeatMessage MUST use a random padding of at
   least 16 bytes.  The padding of a received HeartbeatMessage message
   MUST be ignored.

   If the payload_length of a received HeartbeatMessage is too large,
   the received HeartbeatMessage MUST be discarded silently.
```

Figure 2: The section of RFC 6520 that describes the critical length check

Even without semantic contradictions, RFCs are often ambiguous and a source of misinterpretation. RFC 2616, the original HTTP/1.1 specification, has recently been replaced with six new RFCs (7230-7235), which do not add any new functionality, but simply rephrase the original standard to minimize the **misinterpretation and ambiguity** problems. This process took seven years. Most RFCs do not receive this much attention.

The information about the critical bounds check that led to Heartbleed is scattered over the message format specification in RFC 6520 (see Figure 2). The actual formula is hidden in the description of the padding field and has to be solved for payload_length for the implementation. In this particular case, the co-author of the RFC and the programmer of the OpenSSL implementation are the same person, so this is probably not a problem of text comprehension, but it proves that highlighting potential security problems was not a priority in the standardization process. The lack of **formalism** in RFCs culminates in the absence of standardized tests, which could be used to check the standard conformance of an implementation.

The biggest problem of DTLS/TLS is probably its **feature bloat**. Despite its critical nature as one of the internet's core security components and the consequential need for conciseness, the protocol suite overflows with extensions. Wikipedia currently lists 42 RFCs covering TLS (see Figure 3). This list is not comprehensive. For example, it does not include RFC 6520 (heartbeats). A complete implementation of TLS must cover all these standards, even the outdated ones, because of the need of backwards compatibility with legacy clients and servers. The enormous complexity of the standard does however hinder the widespread adoption of the current version. DTLS/TLS does not adhere to the Unix philosophy: "Write programs that do one thing and do it well", which is of special importance of security critical components, which need to minimize their attack surface.

Finally, the heartbeat extension did introduce unnecessary complexity that neither adds functionality nor security. The 16 byte minimum padding is not justified anywhere in the RFC. The PhD thesis of one of the co-authors states that "without additional random bytes this payload can be guessed easily, thus making it prone to a Known-Plaintext Attack (KPA)" [Seg12], because heartbeat implementations may use simple counters as the request payload. Modern ciphers like AES are specifically designed to be immune to KPAs, and even if a weakness that makes KPAs realistic would be found, the DTLS heartbeats would be our least concerns. Countless applications encrypt easy-to-guess plaintexts with the ciphers available in DTLS/TLS, especially all the standard internet protocols like HTTP, FTP, and SMTP, which are encapsulated in TLS. If TLS does not add random padding in general to prevent attacks, there is no point adding them in a rarely-used extension.

This issue points to a deeper problem: **cargo cult security**. Computer security and cryptography are highly complex and demanding fields, and any small mistake may jeopardize the overall security. Nonetheless, security-critical components are often designed and implemented by system programmers with little or no background in security. They mimic the methods of security experts without fully understanding the underlying principles, much like the cults of oceanic cultures who tried to mimic the "rituals" of Japanese and Allied military to attract airplanes dropping valuable cargo. We need to strengthen the ties between the systems and security communities to improve the mutual understanding.

Overall, one must consider DTLS/TLS as a failed protocol. Its complexity, contradictions, and legacy support requirements make it impossible to fix. New transport protocols like CUSP [TLLB10] or QUIC [Ros13] consequently integrate security into their own functionality instead of interfacing with TLS. Any such attempt must minimize the complexity of the protocol and provide a coherent specification. Formal methods for protocol design, Protocol versions from SSL 2.0 to TLS 1.2:

- Hickman, Kipp E.B. ”The SSL Protocol”. (Internet Draft)
- RFC 6101: ”The Secure Sockets Layer (SSL) Protocol Version 3.0”.
- RFC 2246: ”The TLS Protocol Version 1.0”.
- RFC 4346: ”The Transport Layer Security (TLS) Protocol Version 1.1”.
- RFC 5246: ”The Transport Layer Security (TLS) Protocol Version 1.2”.

Extensions to TLS 1.0 include:

- RFC 2595: ”Using TLS with IMAP, POP3 and ACAP”.
- RFC 2712: ”Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)”.
- RFC 2817: ”Upgrading to TLS Within HTTP/1.1”.
- RFC 2818: ”HTTP Over TLS”.
- RFC 3207: ”SMTP Service Extension for Secure SMTP over Transport Layer Security”.
- RFC 3268: ”AES Ciphersuites for TLS”.
- RFC 3546: ”Transport Layer Security (TLS) Extensions”.
- RFC 3749: ”Transport Layer Security Protocol Compression Methods”.
- RFC 3943: ”Transport Layer Security (TLS) Protocol Compression Using Lempel-Ziv-Stac (LZS)”.
- RFC 4132: ”Addition of Camellia Cipher Suites to Transport Layer Security (TLS)”.
- RFC 4162: ”Addition of SEED Cipher Suites to Transport Layer Security (TLS)”.
- RFC 4217: ”Securing FTP with TLS”.
- RFC 4279: ”Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)”.

Extensions to TLS 1.1 include:

- RFC 4347: ”Datagram Transport Layer Security”.
- RFC 4366: ”Transport Layer Security (TLS) Extensions”.
- RFC 4492: ”Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)”.
- RFC 4680: ”TLS Handshake Message for Supplemental Data”.
- RFC 4681: ”TLS User Mapping Extension”.
- RFC 4785: ”Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS)”.
- RFC 5054: ”Using the Secure Remote Password (SRP) Protocol for TLS Authentication”.
- RFC 5077: ”Transport Layer Security (TLS) Session Resumption without Server-Side State”.
- RFC 5081: ”Using OpenPGP Keys for Transport Layer Security (TLS) Authentication”.

Extensions to TLS 1.2 include:

- RFC 5288: ”AES Galois Counter Mode (GCM) Cipher Suites for TLS”.
- RFC 5289: ”TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)”.
- RFC 5746: ”Transport Layer Security (TLS) Renegotiation Indication Extension”.
- RFC 5878: ”Transport Layer Security (TLS) Authorization Extensions”.
- RFC 5932: ”Camellia Cipher Suites for TLS”.
- RFC 6066: ”Transport Layer Security (TLS) Extensions: Extension Definitions”.
- RFC 6091: ”Using OpenPGP Keys for Transport Layer Security (TLS) Authentication”.
- RFC 6176: ”Prohibiting Secure Sockets Layer (SSL) Version 2.0”.
- RFC 6209: ”Addition of the ARIA Cipher Suites to Transport Layer Security (TLS)”.
- RFC 6347: ”Datagram Transport Layer Security Version 1.2”.
- RFC 6367: ”Addition of the Camellia Cipher Suites to Transport Layer Security (TLS)”.
- RFC 6460: ”Suite B Profile for Transport Layer Security (TLS)”.
- RFC 6655: ”AES-CCM Cipher Suites for Transport Layer Security (TLS)”.
- RFC 7027: ”Elliptic Curve Cryptography (ECC) Brainpool Curves for Transport Layer Security (TLS)”.

Encapsulations of TLS include:

- RFC 5216: ”The EAP-TLS Authentication Protocol”

Figure 3: List of relevant RFC from the Wikipedia article on Transport Layer Security

like language security based approaches [PB11], and a more sophisticated standardization model than traditional RFCs should be considered in this context.

3.3 Computer Science Culture

The internet may be the most outstanding success of academic computer science research. Within a single generation, basic research turned into a world-spanning infrastructure that is ubiquitous to virtually every aspect of modern society. When claiming its success, one must also take responsibility for its failures. Like many other internet protocols, the heartbeat extension is the result of an academic research project. This is no coincidence and our academic culture actually promotes the problematic practice behind it.

First of all, the **publish or perish** culture encourages to produce more instead of better results. A PhD thesis that yielded only a handful or even no paper at all is commonly considered a failure. On the other hand, the peer review process is not sufficient to ensure a minimum quality. To put it bluntly, any programming project paper can be published at some workshop as long as it comes with a few performance figures. Making matters worse, the practicality of an approach is often "proven" by patching popular open-source implementations and submitting the patch to the upstream maintainers. Responsible peer review cannot be outsourced to volunteer code reviewers from the open-source community.

While publishing positive results is too easy, publishing **negative results** is much harder[1]. Papers with a "we did not do X and here is why" approach are rare, especially at prestigious conferences. This is fatal in two ways. First, it encourages researchers to focus on building systems instead of reflecting on the underlying design principles. Second, it impedes the distribution of knowledge about fallacies, dead ends, and pitfalls.

Another problem roots in the success of the internet. It is hard to argue against the success stories of TCP or TLS. Many experts consider the transport layer a "done thing". Only minor tweaks seem practical, a paradigm shift in contrast utopian. This **conservation culture** slows down innovation in the field.

However, giving up completely on the existing infrastructure and building **castles in the air** with clean-slate internet architectures that are not practical does not help the situation either. Any realistic solution must be incrementally deployable on the current internet infrastructure. But that is actually less difficult than skeptics make it seem.

4 An Alternative Heartbeat Proposal

While I personally believe that DTLS/TLS is broken beyond repair[2] and that they should be abandoned as soon as possible, so that all energy can be focused on the design of a sane replacement, I will take the time to sketch an alternative design for DTLS heartbeats to show that it would have been simple to reduce the attack surface. That said, I believe that there is no place for keep-alive and PMTUD functionality in transport security. Those features exclusively belong to the traditional transport layer. If you need them in DTLS/TLS, it only indicates that you are doing something fundamentally wrong and should start over with your design.

First of all, the **redundant functionality** of the type field in the record header and inside the individual record type is superfluous. Using the type field in the record header exclusively as a "protocol discriminator" is completely unnecessary. No security-critical protocol should need to use more than the 256 different message types that fit in the one

[1]This paper being the exception to the rule, of course.

[2]The same goes for many of the associated technologies like ASN.1 and concepts like certificate authorities.

byte type field, because many message types imply high complexity. Actually, TLS uses just 50 message types (which already is a lot), even if all 30 different alerts are counted as individual types [IAN14].

Secondly, the design should observe the **separation of concerns**. Requests and responses, keep-alive and PMTUD messages should not share the same message type and format. Aside from simplifying the message layout and message handlers, this allows to selectively turn these features on or off as needed. A bug in an unneeded and thus disabled extension would neither impact security nor functionality of actually needed extensions.

Finally, **variable-length fields** should be avoided wherever possible. Without a length field, there is no possibility of a missing or wrong boundary check. In TLS, we are stuck with the length field in the record header, but we should not add additional length fields without need. For keep-alive and PMTUD, a simple fixed-length identifier is sufficient to map responses to outstanding requests.

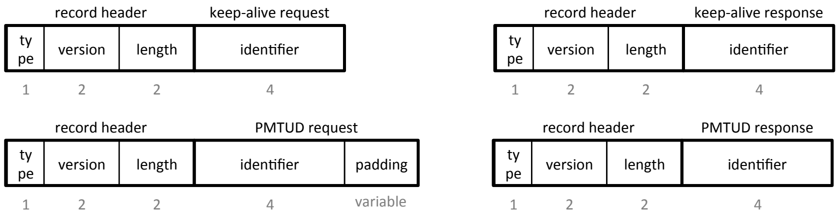


Figure 4: Alternative heartbeat message format

This leaves us with four simple message types (see Figure 4), each of them with a clear purpose and concise specification. The payloads of a keep-alive request and a keep-alive response each consist of a 4 byte identifier. A receiver only needs to check if the record content has a length of four. Even if this check was absent, an attacker could read just four bytes of random memory from the target (instead of 65,535 bytes).

The path MTU discovery needs minimal extra functionality in the form of a padding field in the request. The content of this field is irrelevant and only there to test the routers on the path. The receiver only reads and echoes the identifier. The only change in the handler is to assert a length of at least four instead of exactly four.

A specific message format for the hypothetical mobility extension seems unnecessary. As proposed in the original design, the client can use normal keep-alive messages to make address changes visible to the server. In the case of an unexpected sender address, the server can simply reply with an alert message notifying the client of the change. The server does not need to keep any state. The client can then prove its identity by initiating a TLS session resumption [SZET08]. By going through a connection reestablishment with the same master secret, the possession of the credentials can be validated by the server. If the necessary functionality is already in the standard, it is unnecessary to reinvent it.

5 Dealing with Heartbleed Attacks

Heartbleed attacks are especially dangerous because they leave no trace. An intrusion detection system (IDS) can be taught to detect Heartbleed attacks reliably [Ama14], but that helps only against future attacks. Therefore, it is practically impossible to decide if the vulnerability has been abused before its public disclosure.

The only method to know for sure is network forensics on traffic dumps of the complete timespan that the vulnerability was available. Practically no one keeps comprehensive traffic dumps for over two years (if at all). At ICSI, we keep one to two months of traffic dumps for research purposes. The Lawrence Berkeley National Lab (LBNL) keeps up to three months. With these and the updated IDS scripts, a large number of attacks can be detected after the public disclosure, but none in the time before (see Figure 5). It has to be noted that many of the detected attacks are likely to be deliberate self-tests by the system administrators.

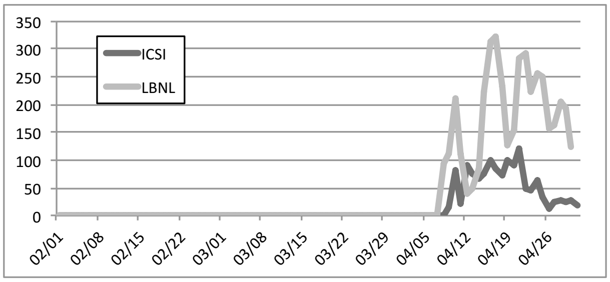


Figure 5: Detected Heartbleed attacks at ICSI and LBNL

The result only shows that the two networks were not attacked within the given timeframe. An earlier attack may have gone undetected. Also, there might have been specifically tailored attacks against other organizations. Without comprehensive traffic dumps covering the complete attack window and all possible targets, the only thing we can conclude with reasonable certainty is that the vulnerability was not used for random 'carpet bombing' attacks recently.

Network forensics is a powerful tool in this context, but it is also a dangerous one. In order to be able to detect and analyze attacks post-mortem, one must keep large amounts of potentially sensitive communication data over a long period of time. This information may become a viable target of attacks itself and thus must be secured carefully. Many users are unable to provide the necessary security and administration locally. Storing the communication data at a campus level or centrally at an internet service provider does aggravate the attack potential of the data and increases the risk of unwanted eavesdroppers.

Furthermore, the forensic analysis is only possible, because the attacks are rather simple and the confidentiality of TLS is limited (record headers are never encrypted). A better transport security or a more complex vulnerability could make forensic analysis of traffic dumps impossible. Surrendering encryption in favor of network forensics is certainly not the right answer. There currently is no common interface that allows to capture the traffic before encryption either, and implementing one would create a dangerous new attack surface. Therefore, the ability to detect and analyze such attacks may diminish further in the future, which emphasizes the importance of a paradigm shift in protocol engineering and implementation.

6 Conclusion

Heartbleed is not just a programming error. It is a chain reaction of failures in the current way of designing and building critical internet protocols. A bug as bad as Heartbleed or even worse can happen again anytime, and it will if we do not change the culture of researching, designing, and implementing internet protocols. Because the internet has become a crucial infrastructure of the global society, it would be irresponsible not to take action.

We need to apply state-of-the-art software engineering methods and tools, need better written and more concise specifications, and should acknowledge that practical computer science is not about building everything that is possible but about understanding how to build good systems. In summary, we need to put security and reliability first and break with bad habits.

Acknowledgments

I would like to thank Johanna Amann for the measurement of Heartbleed attacks in ICSI and LNBL datasets.

References

[Ama14] Johanna Amann. Detecting the Heartbleed Bug Using Bro. `http://blog.bro.org/2014/04/detecting-heartbleed-bug-using-bro.html`, April 2014.

[And72] James P Anderson. Computer Security Technology Planning Study. Volume 2. Technical report, DTIC Document, 1972.

[Bec14] Bob Beck. LibreSSL - An OpenSSL Replacement. `http://www.libressl.org`, 2014.

[Bla14] Mike Bland. Call for OpenSSL Testing Help. `http://mike-bland.com/2014/06/02/call-for-openssl-testing-help.html`, June 2014.

[Die08] Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008.

[Hic95] Kipp Hickman. The SSL protocol. Internet Draft, April 1995.

[IAN14] IANA. Transport Layer Security (TLS) Parameters. `http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml`, June 2014.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

[Ohl14] Ohloh Black Duck Open Hub. The OpenSSL Open Source Project. `http://www.ohloh.net/p/openssl`, 2014.

[PB11] Meredith L. Patterson and Sergey Bratus. The Science of Insecurity. 28c3 presentation, December 2011.

[RM12] Eric Rescorla and Nagendra Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, January 2012.

[Ros13] Jim Roskind. QUIC: Quick UDP Internet Connections. `https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/`, December 2013.

[Seg12] Robin Seggelmann. *SCTP: Strategies to Secure End-To-End Communication*. PhD thesis, Universität Duisburg-Essen, October 2012.

[SZET08] Joseph Salowey, Hao Zhou, Pasi Eronen, and Hannes Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 4507, January 2008.

[The14] The Linux Foundation. Core Infrastructure Initiative. `http://www.linuxfoundation.org/programs/core-infrastructure-initiative`, April 2014.

[TLLB10] Wesley W. Terpstra, Christof Leng, Max Lehn, and Alejandro P. Buchmann. Channel-based Unidirectional Stream Protocol (CUSP). In *Proceedings of IEEE INFOCOM'10*, March 2010.

[TSW12] Michael Tuexen, Robin Seggelmann, and Michael Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520, February 2012.

[Una14] Ted Unangst. Analysis of OpenSSL Freelist Reuse. `http://www.tedunangst.com/flak/post/analysis-of-openssl-freelist-reuse`, April 2014.